Project Integration Architecture: Initial Plan for Distributed User Authentication and Access Control

Dr. William Henry Jones
National Aeronautics and Space Administration
John H. Glenn Research Center at Lewis Field
Cleveland, OH 44135
216-433-5862
William.H.Jones@grc.nasa.gov

```
X00.02  17 May 2001
X00.01  08 May 2001
X00.00  02 May 2001
```

Keywords:
PIA; PRICE; Application Integration; Distributed Lock Managment; Security; Distributed User Authentication; Distributed Access Controls; Trusted Servers

ABSTRACT: *The Project Integration Architecture (PIA) application wrapping architecture is being migrated from its current demonstration implementation in the C++ language to a distributed, net-accessible implementation based upon the Common Object Request Broker Architecture (CORBA). It is explicitly assumed that in this distributed form, multiple users and multiple simultaneous accessors of individual objects, their methods, and their information will be an unavoidable fact of life. This paper develops the initial strategy for assuring the security and integrity of objects and their information in such an environment.*

1 Introduction

In the late 1980's, the Integrated CFD and Experiments (ICE) project [1, 2] was carried out with the goal of providing a single, graphical user interface (GUI) and data management environment for a variety of CFD codes and related experimental data. The intent of the ICE project was to ease the difficulties of interacting with and intermingling these disparate information sources. The project was a success on a research basis; however, on review it was deemed inappropriate, due to various technical limitations, to advance the effort beyond the successes achieved.

A re-engineering of the project was initiated in 1996 [3]. The effort was first renamed the Portable, Redesigned Integrated CFD and Experments (PRICE) project and then, as the wide applicability of the concepts came to be appreciated, the Project Integration Architecture (PIA) project. The provision of a GUI as a project product was eliminated and attention was focused upon the application wrapping architecture. During the intervening years, work has proceeded and an operational demonstration of the PIA project in a C++, single-machine implementation has been achieved.

1.1 Task

The next phase of work focuses on the migration of the PIA Application Classes into a net-enabled, net-accessible, distributed-object architecture. In this vision, each wrapped application would be served out onto the net (intended ultimately as being the global internet) by a server program. In some instances, two or more applications might be served by a single program; however, in most instances geographically dispersed servers dedicated to single applications are envisioned.

This vision brings with it the unavoidable consequence of multiple users and simultaneous access. The broad goal, then, is to devise mechanisms to assure the integrity of individual objects, the information they contain, and the structures of which they are members in such a multiple-user, mutliple access environment.

A base has been laid for this task with the migration of the PIA Foundation Classes to the Common Object Request Broker Architecture (CORBA). A distributed, per-object lock mechanism [4] has been devised which assures information integrity in a multiple access environment. The key interfaces involved in this base are as follows.

1. **GObjLck**: A lockable base interface. This interface recognizes the concept of locking an instance for operational purposes. The vast majority of PIA foundation interfaces are derived from this interface.

2. **GLockCtx**: A context within which locks are held. Each logical thread of execution acquires its own **GLockCtx** instance which holds the various locks necessary to perform a specific operation. The implementation also provides the intelligence to avoid re-acquiring locks that are already held, thus reducing the lock system overhead.

3. **GLock**: The lock interface, itself. This is the actual lock mechanism. Each lockable instance, typically a derivative of the **GObjLck** interface, acquires an instance of the **GLock** interface to perform the actual locking act. The lock provides a simple grant/deny decision to a lock request, rather than blocking a requestor until the lock can be granted. Further, the lock keeps track of all contexts to which it has granted locks in order to facilitate deadlock detection by a lock context when a requested lock is repeatedly denied.

The key feature of this locking mechanism is that it is distributed just as the objects it locks are distributed. There is one lock for each object and, typically, the lock exists on the same server as the locked object. The limit on simultaneous lock operations is, thus, the same as the limit on servable objects. There is no centralized locking authority to slow down an otherwise distributed, scalable object system.

In this system any **GLockCtx** instance may obtain a lock on any **GLock** instance. There is no concept of differentiation between lock contexts, nor of an access control which would prohibit a particular context from obtaining a lock that another lock context would be granted. This must be added in the implementation of the PIA Application Classes to the CORBA distributed object environment.

Thus, the first goal is to introduce the concept of a user whose identification can be established and, with such an identification, extend the locking paradigm to include the question as to whether or not this user has the right to obtain the requested lock before the actual lock mechanism is operated. This extension, alone, is a relatively simple task.

What makes this first goal interesting is that it brings with it a second goal: establishing that the facts used in processing a lock request are, in fact, true.

2 Security Presumptions

The user authentication and access control mechanism is founded upon two security presumptions.

2.1 Integrity of the Server

It is presumed that the server of an object suite is trustworthy within its software boundaries. That is, an object served by a particular server may expect that another object served by that same physical server operates in accordance with its interface description and the supporting documentation.

This presumption is based upon the expectation that no server would exist for the purpose of hacking itself. A server which could not expect its own served objects to operate correctly would be useless to those creating the server and, thus, would not be created.

2.2 Integrity of Other Servers

It is presumed that a bond of trust can be recorded between two servers. It is presumed that this bond will be established by some non-automatic means, probably between people who decide between themselves that each server is conforming to the appropriate security mechanisms and, thus, can be trusted. This bond is then recorded in each server through some priveleged act.

This bond then can be formed into a graph when multiple servers are involved. If A trusts B and B trusts C, then A trusts C. There is also a recognition that A may trust C somewhat less than it trusts B.

This presumption of trust between servers is somewhat less reasonable than that of trust within a server. It would be possible for a person to lie about the integrity of a server, or for an auditor to miss some cunningly devised flaw, and, thus, for a security breach into the system to be opened. No automated tool to establish the integrity of a server is currently proposed; however, it is possible that such a tool might be devised at some point in the future.

3 Interface Suite

Both derivative and new interfaces must be involved in the user authentication and access control process.

1. **GacBObj**: A **GObjLck** derivative, this basic lockable object introduces the idea that it doesn't necessarily trust the context in which a lock is to be requested.

The interface also provides a connection to the access control description for the instance.

2. **GacDescAccs**: This interface, and its attendant components, provides the data to feed the access control process for a particular object.

3. **GacLockCtx**: A **GLockCtx** derivative, this lock context adds the concept of user identity by providing a reference to a single **GacUser** instance.

4. **GacUser**: This interface identifies a user and, when needed, provides a link back to that user for password (or somesuch) solicitation.

5. **GacLock**: A **GLock** derivative, this lock form understands that an access control list must be processed to determine whether the requesting lock context is entitled to hold the requested lock before any determination as to whether or not that lock can be granted is made.

6. **GacSrvrCtl**: A **GSrvrCtl** derivative, this server object must establish the validity of the **GacLockCtx** instance and the user it identifies.

4 The Process

The following subsections discuss the expected flow of events to grant access to an instance of a **GacBObj**-derived interface.

4.1 Lock Process Initiation

An instance, following the form established in the foundation classes, is responsible for locking itself. Thus, a method, upon invokation, must obtain the appropriate lock for the intended operation before that operation is begun in substance. (In some cases, modulation of the locking state throughout the operation may occur, but this can be considered as the operation of multiple interior methods within a containing method and, thus, does not impact the present discussion.)

In the established locking architecture, the lock request was made by the **GObjLck** interface not to the associated **GLock** instance directly, but to the **GLockCtx** lock context with a reference to the appropriate **GLock** instance provided. This was done to allow the **GLockCtx** instance to determine whether it might already hold the required lock and, thus, short circuit the overhead of the lock mechanism. (Multiple locking of the same instance is likely as methods invoke methods on the same instance to perform component tasks.) The consequence of this, though, is that it is the **GLockCtx** instance which returns to the method the information that the needed lock has, in fact, been obtained.

The wrinkle to the basic system that **GacBObj** must now add is a fundamental distrust of the lock context object. Since it is the lock context object which returns the final disposition of the lock action, a corrupted lock context object could be devised and provided that would report the granting of locks when those locks had, in fact, been denied by or, more likely, never even requested of the **GacLock** instance. Thus, **GacBObj** must be coded to ask the 'system', in our scheme the **GacSrvrCtl** instance, to establish the integrity of the requesting **GacLockCtx** instance before initiating the lock process by a call to that lock context instance.

4.2 Lock Context Validation

The **GacSrvrCtl** interface (of which there is to be exactly one instance for each server program) has the job of determining the validity of a **GacLockCtx** instance intended to carry out a lock act.

In the first wack at this task element, it was proposed that a lock context instance be considered to be valid if it was either

1. Served by the server serving the validating **GacSrvrCtl**, or

2. Served by a server reachable through the trusted server graph of which the validating **GacSrvrCtl** instance is a member, that server being reached directly, indirectly, or indirectly through a path of no more than a specified length.

The use of the **islocal** functionality provided by all CORBA objects was considered inappropriate since a lock context designed to breach the security system could certainly respond falsely to this method, as indeed it could to virtually any method.

This first proposed security standard has been discarded when it was realized that a corrupt lock context could tie itself to an uncorrupt lock context for the purposes of satisfying security tests (which would be satisfied by the tied lock context) while substituting corrupt lock actions when those were invoked.

Thus, the revised scheme is a simple, operational one. A lock context instance to be validated will simply be exercised by the validating **GacSrvrCtl** instance in attempts to lock and release various system-provided lock targets whose responses are known to the testing **GacSrvrCtl**. A certain randomness will be introduced into this testing so as to eliminate the possibility of a corrupt lock context being developed to always operate to standards for, say, the first lock attempt, and the corruptly thereafter.

Furthering this new proposed validation method, the graph of trusted servers will not be utilized for this validation. In this way, further randomness will be introduced into the validation process since it will be difficult to predict when a lock on some other server might be requested, causing a revalidation of the lock context.

The lock targets of this validation process are to be **GacLock** instances whose lock request responses have been forced to known conditions programmed by the utilizing **GacSrvrCtl** instance. Such programmed responses need to be a protected function of the **GacLock** interface so that a corrupt lock context tied to a valid lock context may not switch to its tied accomplice when such a lock is encountered as its target.

To further meet anticipated paranoia, the test lock(s) should be connected back to the instance requesting the lock validation. This will disallow the possibility of a corrupt lock context keying its operation (through the **GetLockedObject** method of **GLock**) to the instance it was intended to corruptly lock. To further circumvent this attack, it is important for **GacBObj**-based interfaces not to provide the ability to identify their controlling lock instances.

This last effort suggests that the **GacSrvrCtl** will want to keep a pool of test lock instances readily at hand so that an individual instance may be devoted to testing a individual lock context.

4.2.1 Cacheing Validated Lock Contexts

In order to short circuit the time consuming task of a complete validation search for each locking act (which might be repeated many times for a single method invokation from an external source), the **GacSrvrCtl** will doubtless want to maintain a cache of previously validated lock context instances. For reasons that will be explained later, the cache will want to associate the validated lock context object with the final user identification referenced by it, so a **GMapGObjToGObj**-based implementation suggests itself for this cache.

In the event that validation involved a traversal of the trusted server graph, it is probably appropriate for each server control in the validating traversal to capture the validated lock context in its own cache. That is, if A inquired of B, who inquired of C, who inquired of D, who confirmed that the subject lock context instance was valid, then D, C, and B, in addition to A, should probably all capture the subject lock context in their validated lock context caches. It may well be that B, C, and D will never need to validate that lock context for themselves (that is, for operations within their server), but the expense of validation might well outweigh the cost of capturing a validation that has already succeeded.

Because the **GacSrvrCtl** object(s) will be maintaining a reference to the validated lock context, it will be necessary for the implementation of the **GacLockCtx** interface to maintain a map of validating **GacSrvrCtl** instances and inform those instances when the lock context passes out of existence. In this way validated lock context cache bloat may be avoided.

The cache may wish to time-stamp entries for various purposes.

1. A purge of the cache based on age may be appropriate to keep the number of entries down and to remove validated lock contexts that have simply disappeared from use without informing the validating **GacSrvrCtl**.
2. It may be appropriate to periodically re-validate lock contexts based on age. This is not planned for the first implementation, but a security attack by exchanging lock context implementations after validation might be remotely possible.

4.2.2 Lock Context for Validation Operations

In order not to be a bottleneck, the lock context validation operation will, of course, operate in multi-threaded server mode, just as all other method invokations do. This presents a complication in that the structures used, in particular the cache of previously validated lock contexts, will have to be locked. This will require a lock context to hold the acquired locks and, since the subject lock context is yet to be trusted, it cannot hold those locks. Furthermore, the subject lock context might not have sufficient privelege to hold such locks (although privelege may be a concept that should be dispensed with for the sake of speed in the light of the closely held nature of the structures involved). Finally, the subject lock context ought not to be bolloxed up with a mess of locks not of interest to the thread of execution that it represents.

Thus, a series of ‘system’ lock contexts will be needed for the use of the **GacSrvrCtl** instance in this effort. While these might be created on the fly, it will probably be more effective to maintain a pool of these lock contexts (one per server thread, or perhaps a few more?) and devise a mechanism by which a thread can quickly acquire one.

Additionally, since these ‘system’ lock contexts will (or may) be used in locking objects that know only the standard locking system, it will be necessary for them (the lock contexts) to pass through the same validation process. Because of their high frequency of use, a quicker cache dedicated to these contexts may be appropriate.

4.3 User Identification

Having received assurance from the governing **GacSrvrCtl** instance that the identified **GacLockCtx** object is trustworthy, the **GacBObj**-based instance then invokes **RequestLock** in presentation by that lock context instance. At present, no altered behavior in that functionality is anticipated and it, after assuring that the desired lock is not already in its possession, further invokes **RequestLock** in presentation by the identified **GLock**-based lock instance, presumably that being a **GacLock** instance.

Overriding **RequestLock** functionality in **GacLock** must first determine whether access controls exist for the controlled object. If such controls do exist, they are to be processed before the basic lock mechanisms are operated. A user identification is required for the operation of this process.

The matter of user identification (other than user ‘root’) is of some import. The derived **GacLockCtx** interface is to provide a reference to a single, corresponding **GacUser** interface instance. The **GacUser** instance may (or may not) carry along a text string (perhaps encapsulated in a **GString** object) identifying the user by name and host name, but it definitely is to provide a call-back function to prompt the (supposed) user for a user identification and password. The retention of user identification by the **GacUser** instance is considered to be optional for the purposes of the system because the lock context and user identification instances are presumed to be under the control of the user/hacker; thus, they could be modified after initial validation/authentication and their persistent contents would not be trusted.

User authentication is to occur only as needed, that is, when and if an access control hierarchy is to be processed. (There may be some cases in which a lock context serves without ever requesting access to a controlled instance.) At the point that the **GacLock** instance determines that it will process an access control hierarchy, it is to go back to its validating **GacSrvrCtl** instance and ask for the user identification associated with the lock context.

If no user identification has been associated with the lock context, the authentication call-back function is invoked in presentation by the **GacUser** instance to obtain a user identification and password. The user identification is to include the full host name of the user’s machine, for example

enjones@witsend.grc.nasa.gov

Such an identifier seems appropriate for network-wide identifications, distinguishing one **enjones** from the next. Because of this, authorized user identification entries will need to be in the form of **PGrep** expressions so as to allow, if desired, users such as **enjones** to log in from any machine at **grc.nasa.gov**. As in the C++ implementation, the grep expression should be coded with care to assure that it does not allow unexpected identifications, such as

enjones@witsend.grc.nasa.gov.redsquare.ruskiland

to get through.

Once the identification and password are in hand, a search is made of authorized user entries in the **GacSrvrCtl** instance. If an appropriate entry does not exist in the presenting **GacSrvrCtl** instance, a traversal of the trusted server graph is to be made in search of a trusted server control that can authenticate the user identification. Whether or not some depth of trust restriction should be placed on this traversal is yet another matter open for discussion.

Assuming that an authentication is obtained, the next step it to enter the resulting user identification in the validated lock context cache. Also, if other servers have cached the validated lock context simply by virtue of being in the trusted server graph path from requester to validator, those other caching servers should be informed of the user identification obtained.

(There may be some issue in trusting a validated lock context to provide a traversal of cacheing servers since that lock context is under the control of the user/hacker; however, at this point no profit can be seen in the possibilities. The malacious user can achieve only two things by adjusting the list of cacheing servers: actual trusted servers may fail to be informed of an authenticated user identification and untrusted servers may be informed of an authenticated user identification. For the first possibility, this means only an increase in overhead in the event that uninformed servers should need the identification in that they would then initiate what is, in fact, a redundant authentication interaction.

For the second possibility, it is difficult to see what advantage can be gained by transmitting to a server the identification (without a password, hashed or otherwise) of an authenticated user. Certainly a malicious user would already know his own identification and have no need to resort to such subtrefuges to learn it.)

4.3.1 User Identification and Password Management

Having established that the **GacSrvrCtl** instance will maintain a user identification/password data base, it will be necessary to provide functions and utilities to access and manage that data base. These operations should be performed only by priveleged users charged with maintaining the server program.

4.4 Access Control List Processing

Having obtained from the resident **GacSrvrCtl** instance a user identification, the **GacLock** instance may then proceed with access control list processing. For the most part, this is a mundane task.

The access control process is one of matching a user identification against an ordered hierarchy of access control entries and applying the access result of the first such matching entry encountered. The access control grants access in modes matching the lock modes: Reference, Read, Write, Execute, and Delete. Release access is granted to all users since the only function of this access is to let go of an object.

The entire access control mechanism should probably be short circuited in the event that the identified user is 'root@local_host', or perhaps just 'root'. This will eliminate the overhead of adding an unlimited-access access control entry for 'root' to every object ever created.

As mentioned above in user identification, the access control list entries will want to be made in terms of a **PGrep** expression so that user identifications (which are, themselves to be exact) can be, in effect, wild carded. This, again, will allow **enjones** to obtain access from any machine ending with a domain of, say, **nasa.gov**.

Another issue to be consider is what to do in the event that no definitive access control determination is made. (This situation arises, really, in only one way: there is no access control entry matching the identified user. Whether there are entries, but none match, or whether there is simply a complete absense of an access control mechanism is, in fact, beside the point.) It is expected that this situation will be resolved by a characteristic of the controlled **GacBObj** instance indicating whether an indeterminate result should result in complete access or no access. Whether or not there is need to allow for global selections, perhaps solicited for the local **GacSrvrCtl** instance is yet another question open for further discussion.

4.4.1 Controlling Default Access

The default access selection characteristic for an object is, of course, another point of concern. A malicious user that was able to change the default access from none to all would circumvent the entire access control process. Thus, it is important to assure that the setting of the object characteristics be controlled just as all other things in a secured object are. It would be appropriate to override the instance charcteristic functionality to provide the same access control requirements (Execute access, rather than the default Write access, as discussed in the next section) on the default access characteristic as will be applied to the access control list itself.

4.4.2 Controlling the Access Control List

The ability to amend to access control list is also a point of concern. At present, the is no Control lock or access class. It is currently expected that access control list adjustment operations will be performed under an Execute lock, which assures exclusive access to the object and denies control priveleges to those merely allowed to change the object data (Write access).

If it is determined that the finer access resolution is needed, it is a relatively small matter to add an Access lock level to the system. This involves only the introduction of the symbolic definition and the expansion of the **GLock** state table from a 6-by-6 matrix to a 7-by-7.

4.4.3 Access Control List Hierarchy

The access control list is encapsulated through the **GacDescAccs** interface and its components. This is a descriptive element interface and, as such, is attached to a **GacBObj** instance through the layered descriptive system. The normal order of search is from the shallowest descriptive layer to the deepest.

Should a search of attached access descriptions for a **GacBObj** instance fail to produce a definitive answer, the possibility exists that the search could be continued through the uplink chain implemented by the **GacBObj** interface. Thus, the instance defining the containing structure (for example the **GacCfg** containing a **GacPara** instance, and then the **GacAppl** containing that **GacCfg**) would be searched. By this means, access to broad groups of objects could be controlled by single access control lists.

4.4.4 Access Control List Management

Having established that **GacBObj**-based instances will maintain an access control data base, it will be necessary to provide functions and utilities to access and manage that data base. These operations should be performed only by priveleged users charged with maintaining the served application. Probably, these will be the same as those that maintain the server, but it is possible for these users to be distinct sets.

4.5 Lock Mechanism Operation

Presuming the access control hierarchy processing has determined that the identified user is permitted to obtain the kind of lock desired, the **RequestLock** functionality inherited from the **GLock** base interface is now invoked to determine whether or not the present conditions allow the granting of the lock. The grant/deny decision is then transmitted back through the **GacLockCtx** instance to the originating **GacBObj** method which then proceeds with operations, or aborts them, as appropriate.

4.6 Access Event Notification

While the **GacLock** interface itself is not a derivative of the **GacBObj** class it will, customarily, be locking instances of that interface and, through the **IsKindOfInterface** mechanism it can be determined when this is, in fact, the case. The **GacBObj** interface will, of course, support the event mechanism prototyped in the **PacBObj** and **PacEvent** classes. By combining these capabilities, it is possible to implement an access event mechanism, allowing monitoring and other facilities to be informed when access is granted and, more importantly, when it is denied.

When this access event mechanism is implemented, it will be necessary to provide a mechanism to establish the ultimate destination as being trusted. The exact nature of this determination is yet to be established. It may be that a special access control lock will be sufficient for a first attempt in this area.

The reason for such security precautions is simply that the stream of access denial and, particularly, grant events, which will presumably identify authorized and unauthorized users, could be of some use to a user/hacker. In breaking into any system of security, it is usually an important first step to know the identify of those who can be granted access.

5 Depth of Trust Algorithm

The various operations traversing the graph of trusted servers should be devised with a depth limit in their operation. A depth argument should be passed to the visitation function and decremented as the visitation function is re-invoked for the immediate successors of the visited node. Should this depth argument reach 0, further traversal should be terminated and functional return made.

In general, the initial depth of trust will be set to a practical infinity, probably **MAXLONG** or some such. At present, no purpose is identified where a small depth of trust is proposed. It might be valuable for user authentication at some point in the future.

A small inaccuracy arises from the nature of directed graph traversals in implementing the depth of trust concept. Suppose A trusts B, who trusts C and D, and further that C trusts D. The depth first path to D from A is A to B to C to D, giving a depth of trust for D of 4; however, the shortest path to D is A to B to D, giving a depth of trust of 3. In this example, the error is small; however, a graph can certainly be arranged in which the discrepancy is large. Should depth of trust restrictions be imposed, an algorithm to resolve this error would be of considerable interest. Unfortunately, classic graph traversal algorithms do not immediately suggest a solution, especially a quick and effective one. So it goes.

6 General Event Security

There are wider security implications to the event mechanism to be implemented by the **GacBObj** class than simply controlling the stream of access grant and denial events. The event mechanism is intended to provide, at least in some cases, corrective responses to application operations. Thus, a heightened level of security should be applied to those attempts to attach such mechanisms.

While a complex plan of validation and authentication was at first proposed, it is now thought that the basic access control mechanism of the interface system will be sufficient. A user with write, execute, and control access to an object can achieve corruption and malfunction without the necessity of resorting to event mechanism subterfuges. Thus, the application of such access controls to the mechanism attachment process should be sufficient protection in this area.

References

[1] The American Society of Mechanical Engineers. *Integrated CFD and Experiments Real-Time Data Acquisition Development*, number ASME 93-GT-97, 345 E. 47th St., New York, N.Y. 10017, May 1993. Presented at the International Gas Turbine and Aeroengine Congress and Exposition; Cincinnati, Ohio.

[2] James Douglas Stegeman. Integrated CFD and Experiments (ICE): Project Summary. Technical memorandum Number not yet assigned, National Aeronautics and Space Administration, Lewis Research Center, 21000 Brookpark Road, Cleveland, OH 44135, December 2001.

[3] William Henry Jones. Project Integration Architecture: Application Architecture. Draft paper available on central PIA web site, March 1999.

[4] William Henry Jones. Project Integration Architecture: Distributed Lock Management, Deadlock Detection, and Set Iteration. Draft paper available on central PIA web site, April 1999.